2.4 Priority Queues

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation

See video/book/booksite

Priority queue

A collection is a data type that stores a group of items.

<table>
<thead>
<tr>
<th>data type</th>
<th>core operations</th>
<th>data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>Push, Pop</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>queue</td>
<td>Enqueue, Dequeue</td>
<td>linked list, resizing array</td>
</tr>
<tr>
<td>priority queue</td>
<td>insert, Delete-Max</td>
<td>binary heap</td>
</tr>
<tr>
<td>symbol table</td>
<td>Put, Get, Delete</td>
<td>binary search tree, hash table</td>
</tr>
<tr>
<td>set</td>
<td>Add, Contains, Delete</td>
<td>binary search tree, hash table</td>
</tr>
</tbody>
</table>

Collections

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.
Queue. Remove the item least recently added.
Randomized queue. Remove a random item.

Priority queue. Remove the largest (or smallest) item.
Generalizes: stack, queue, randomized queue.

<table>
<thead>
<tr>
<th>operation</th>
<th>arguments</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>E</td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>E</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>E</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>Q</td>
</tr>
</tbody>
</table>
Priority queue API

### Requirement
Items are generic; they must also be Comparable.

Key must be Comparable
(bounded type parameter)

```java
public class MaxPQ<Key extends Comparable<Key>>{
    public MaxPQ(){ create an empty priority queue
    MaxPQ(Key[] a) create a priority queue with given keys
    void insert(Key v) insert a key into the priority queue
    Key delMax() return and remove a largest key
    boolean isEmpty() is the priority queue empty?
    Key max() return a largest key
    int size() number of entries in the priority queue
}
```

**Note.** Duplicate keys allowed; `delMax()` picks any maximum key.

Priority queue: applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Discrete optimization. [bin packing, scheduling]
- Artificial intelligence. [A* search]
- Computer networks. [web cache]
- Operating systems. [load balancing, interrupt handling]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra’s algorithm, Prim’s algorithm]
- Number theory. [sum of powers]
- Spam filtering. [Bayesian spam filter]
- Statistics. [online median in data stream]

Priority queue: client example

**Challenge.** Find the largest \( M \) items in a stream of \( N \) items.
- Fraud detection: isolate \( S \$ \) transactions.
- NSA monitoring: flag most suspicious documents.

\( N \) huge, \( M \) large

**Constraint.** Not enough memory to store \( N \) items.

**Q.** Would you use a MaxPQ or a MinPQ?

Transaction data type is Comparable (ordered by \( S \$ \))

```java
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while (StdIn.hasNextLine()){
    String line = StdIn.readLine();
    Transaction transaction = new Transaction(line);
    pq.insert(transaction);
    if (pq.size() == M) pq.delMin(); // pq now contains largest \( M \) items
}
```

Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 1</td>
<td>P</td>
<td></td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert 2</td>
<td>P Q</td>
<td></td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert 3</td>
<td>P E</td>
<td></td>
<td>3</td>
<td>P E Q E</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>4</td>
<td></td>
<td>4</td>
<td>P E X A</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert 5</td>
<td>P E A M</td>
<td></td>
<td>5</td>
<td>P E X A M</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>6</td>
<td></td>
<td>6</td>
<td>P E M A P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert 7</td>
<td>P E A L M P</td>
<td></td>
<td>7</td>
<td>P E M A P L E</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>8</td>
<td></td>
<td>8</td>
<td>P E M A P L E</td>
<td>A E E L M P P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue.
Priority queue: implementations cost summary

Challenge. Implement all operations efficiently.

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>goal</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

order of growth of running time for priority queue with $N$ items

2.4 Priority Queues

- API and elementary implementations
- binary heaps
- heapsort
- event-driven simulation

Complete binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.

A complete binary tree in nature

Property. Height of complete binary tree with $N$ nodes is $[\log N]$. 
Binary heap: representation

Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.
- Keys in nodes.
- Parent’s key no smaller than children’s keys.

Array representation.
- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!

Binary heap: properties

Proposition. Largest key is \(a[1]\), which is root of binary tree.

Proposition. Can use array indices to move through tree.
- Parent of node at \(k\) is at \(k/2\).
- Children of node at \(k\) are at \(2k\) and \(2k+1\).

Binary heap demo

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

Binary heap demo

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered
**Binary heap demo**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

```plaintext
insert S
```

```
T P R N H O A E I G
```

```
insert S
```

```
T P R N H O A E I G S
```

```
violates heap order (swim up)
```

```
T P R N S O A E I G H
```

```
violates heap order (swim up)
```

```
T S R N P O A E I G H
```

```
T S R N P O A E I G
```

```
violates heap order (swim up)
```

```
T S R N P O A E I G
```

```
violates heap order (swim up)
```

```
T S R N P O A E I G
```
Binary heap demo

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

Binary heap demo

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap demo

insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap demo

insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum
**Binary heap demo**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**remove the maximum**

```plaintext
  H  S  R  N  P  O  A  E  I  G  T
  1  2
```

**Binary heap demo**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**remove the maximum**

```
  V  H  S  P  O  N  R  A
  5
```

**Binary heap demo**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**remove the maximum**

```
  S  H  R  N  P  O  A  E  I  G  T
  1  2
```

**Binary heap demo**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**heap ordered**

```
  S  P  R  N  H  O  A  E  I  G  T
  1  2  5
```
Binary heap: promotion

**Scenario.** A key becomes larger than its parent's key.

**To eliminate the violation:**
- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

**Peter principle.** Node promoted to level of incompetence.

Binary heap: insertion

**Insert.** Add node at end, then swim it up.
**Cost.** At most \(1 + \log N\) compares.

```
public void insert(Key x)
{
    pq[++] = x;
    swim();
}
```

Binary heap: demotion

**Scenario.** A key becomes smaller than one (or both) of its children's.

**To eliminate the violation:**
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++; // why not smaller child?
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

**Power struggle.** Better subordinate promoted.

Binary heap: demotion

**Q.** Write a recursive version of sink

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

This is just an exercise. No particular reason to implement this recursively.

In fact, many compilers will *automatically* convert the recursive version to the iterative one. This is called tail-call elimination or tail-call optimization.
**Binary heap: delete the maximum**

**Delete max.** Exchange root with node at end, then sink it down.

**Cost.** At most $2 \log N$ compares.

```java
public Key delMax() {
    Key max = pq[1];
    exch(1, N--);
    sink(1);
    pq[N+1] = null;
    return max;
}
```

**Priority queue: implementations cost summary**

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

Order-of-growth of running time for priority queue with $N$ items

**Delete-random from a binary heap**

**Problem.** Delete a random key from a binary heap in logarithmic time.

```java
public class MaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;
    private int N;

    public MaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public Key delMax() {
        // see previous code
    }

    private void insert(Key key) {
        // see previous code
    }

    private void delMax(int k) {
        // see previous code
    }

    private void exch(int i, int j) {
        Key t = pq[i];
        pq[i] = pq[j];
        pq[j] = t;
    }

    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    private void swim(int k) {
        // see previous code
    }

    private void sink(int k) {
        // see previous code
    }
}
```
**Delete-random from a binary heap**

**Problem.** Delete a random key from a binary heap in logarithmic time.

**Solution.**
- Pick a random index \( r \) between 1 and \( N \).
- Perform \( \text{exch}(r, N--) \).
- Perform either \( \text{sink}(r) \) or \( \text{swim}(r) \).

**Binary heap: practical improvements**

**Multiway heaps.**
- Complete \( d \)-way tree.
- Parent’s key no smaller than its children’s keys.

**Fact.** Height of complete \( d \)-way tree on \( N \) nodes is \( \sim \log_d N \).

**Priority queue: implementation cost summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Del Max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>( 1 )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>ordered array</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( d )-ary heap</td>
<td>( \log_d N )</td>
<td>( d \log_d N )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>( 1 )</td>
<td>( \log N )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Brodal queue</td>
<td>( 1 )</td>
<td>( \log N )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>impossible</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

\*sweet spot: \( d = 4 \)\*  
\*impossible\*
Binary heap: considerations

Underflow and overflow.
- Underflow: throw exception if deleting from empty PQ.
- Overflow: use resizing array.

Minimum-oriented priority queue.
- Replace `less()` with `greater()`.
- Implement `greater()`.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.

Immutability of keys.
- Assumption: client does not change keys while they’re on the PQ.
- Best practice: use immutable keys.

Immutability: implementing in Java

Immutable data type. Can’t change the data type value once created.

Examples: String, Integer, Double, Color, Vector, Transaction, Point2D.

Mutable: StringBuilder, Stack, Counter, Java array.

To create your own immutable data types:
- Make defensive copy of client-provided mutable variables in constructor
- Don’t change instance variables in instance methods

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

Advantages.
- Simplifies debugging.
- Simplifies concurrent programming.
- More secure in presence of hostile code.
- Safe to use as key in priority queue or symbol table.

Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there’s a very good reason to make them mutable. ... If a class cannot be made immutable, you should still limit its mutability as much as possible.”
— Joshua Bloch (Java architect)
Heapsort

Basic plan for in-place sort.
- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.
- Sortdown: repeatedly remove the maximum key.

Array in arbitrary order

Array in heap order

Heap construction. Build max heap using bottom-up method.

heap (in place)

sorted result
(in place)
Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 5

sink 5

sink 5

sink 5
Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 4

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 4

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3
**Heap construction.** Build max heap using bottom-up method.

sink 3

sink 2

sink 2
Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

sink 1

demo

end of construction phase
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

sink 1
**Heapsort demo**

**Sortdown.** Repeatedly delete the largest remaining item.

**sink 1**

![Heapsort diagram 1](image1)

![Heapsort diagram 2](image2)

**Heapsort demo**

**Sortdown.** Repeatedly delete the largest remaining item.

**exchange 1 and 10**

![Heapsort diagram 3](image3)

![Heapsort diagram 4](image4)
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10

sink 1

sink 1

sink 1
Heap sort demo

**Sortdown.** Repeatedly delete the largest remaining item.

```
  S
 /\ \
 P  R
/ \ / \ \
O  L  E  A \\
M  T  X
```

**Result (heap-ordered)**

```
SPROLEAMETX
```

Heap sort demo  

**Sortdown.** Repeatedly delete the largest remaining item.

```
A
  E
  E
  L M O P
  R S T X
```

array in sorted order

```
A E E L M O P R S T X
```

Heap sort: heap construction

**First pass.** Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)
  sink(a, k, N);
```

Heap sort: sortdown

**Second pass.**

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
  
  exch(a, 1, N--);
  sink(a, 1, N);
```

```
A
  E
  E
  L M O P
  R S T X
```

```
A E E L M O P R S T X
```

```
A E E L M O P R S T X
```

```
A E E L M O P R S T X
```
Heapsort: Java implementation

```java
public class Heap {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int k = N/2; k >= 1; k--) {
            sink(a, k);
        }
        while (N > 1) {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }

    private static void sink(Comparable[] a, int k, int N) {
        if (k >= N) return;
        exch(a, k, N);
        sink(a, (int)(k/2), --N);
    }

    private static int parent(int j) {
        return j/2;
    }

    private static void exch(Object[] a, int i, int j) {
        Object temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    // but make static (and pass arguments)
    private static void sink(Comparable[] a, int k, int N) {
        // as before */
    }

    private static boolean less(Comparable[] a, int i, int j) {
        // as before */
    }

    private static void exch(Comparable[] a, int i, int j) {
        // as before */
    }

    private static int less(Comparable[] a, int i, int j) {
        return 0;
    }
}
```

but make static (and pass arguments)
private static void sink(Comparable[] a, int k, int N) {
    // as before */
}

private static boolean less(Comparable[] a, int i, int j) {
    // as before */
}

private static void exch(Comparable[] a, int i, int j) {
    // as before */
}

Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>SORT L X A M P E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT L X A M P E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SORT X T L A M P E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S T X P L R A M O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X T S P L R A M O E E</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>T P S O L R A M E X</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>S P R O L E A M E T X</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>R P E O L E A R S T X</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>P D E M L E A R S T X</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>O M E A L E P R S T X</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>M L E A E D P R S T X</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>L E A M O P R S T X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E A E M O P R S T X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>E A E M O P R S T X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A E E L M O P R S T X</td>
</tr>
</tbody>
</table>

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

Proposition. Heap construction uses \( \leq 2N \) compares and \( \leq N \) exchanges.
Proposition. Heapsort uses \( \leq 2N \lg N \) compares and exchanges.

\[
N \log N \text{ algorithm can be improved to } \approx 1 \lg N \text{ (but no such variant is known to be practical)}
\]

Significance. Deterministic in-place \( N \log N \) sorting algorithm.
- Mergesort: no, linear extra space.
- Quicksort: no, randomized.
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:
- Inner loop longer than quicksort’s.
- Makes poor use of cache.
- Not stable.

algorithm can be improved using advanced caching tricks
Sorting algorithms: summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>$\frac{1}{2} N^2$</td>
<td>$N^2$</td>
<td>$\frac{3}{2} N^2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️ ✔️</td>
<td>$N$</td>
<td>$\frac{1}{4} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>merge</td>
<td>✔️ $\frac{1}{3} N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee; stable</td>
<td></td>
</tr>
<tr>
<td>timsort</td>
<td>✔️ $N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️ $N \lg N$</td>
<td>$2 N \lg N$ (expected)</td>
<td>$\frac{1}{2} N^2$</td>
<td>$N \log N$</td>
<td>probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️ $N$</td>
<td>$2 N \lg N$ (expected)</td>
<td>$\frac{1}{2} N^2$</td>
<td>$N \log N$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>heap</td>
<td>✔️ $3 N$</td>
<td>$2 N \lg N$</td>
<td>$2 N \lg N$</td>
<td>$N \log N$</td>
<td>guarantee; in-place</td>
</tr>
<tr>
<td>?</td>
<td>✔️ ✔️ $N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.

Hard disc model.
- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.

2.4 Priority Queues

- API and elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation

Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.

Hard disc model.
- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.
Warmup: bouncing balls

**Time-driven simulation.** \( N \) bouncing balls in the unit square.

```java
public class BouncingBalls {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        Ball[] balls = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while(true) {
            StdDraw.clear();
            for (int i = 0; i < N; i++)
                balls[i].move(0.5);
            StdDraw.show();
        }
    }
}
```

### Main simulation loop

- Discretize time in quanta of size \( dt \).
- Update the position of each particle after every \( dt \) units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

**Fundamental challenge for time-driven simulation**

- If \( dt \) is too small: excessive computation.
- If \( dt \) is too large: may miss collisions.

---

- Physics problems: when? what effect?
- CS problems: which object does the check? too many checks?

---

**Time-driven simulation**

**Main drawbacks.**

- \( \sim N^2/2 \) overlap checks per time quantum.
- Simulation is too slow if \( dt \) is very small.
- May miss collisions if \( dt \) is too large.

(if colliding particles fail to overlap when we are looking)
### Event-driven simulation

Change state only when something interesting happens.
- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain PQ of collision events, prioritized by time.
- Delete min = get next collision.

**Collision prediction.** Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

**Collision resolution.** If collision occurs, update colliding particle(s) according to laws of elastic collisions.

### Particle-wall collision

**Collision prediction and resolution.**
- Particle of radius \( r \) at position \((x, y)\).
- Particle moving in unit box with velocity \((v_x, v_y)\).
- Will it collide with a vertical wall? If so, when?

\[
\Delta t = \begin{cases} 
\infty & \text{if } \Delta v \cdot \Delta r \geq 0, \\
\infty & \text{if } d < 0, \\
\frac{-\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta r} & \text{otherwise}
\end{cases}
\]

\[
d = (\Delta x \cdot \Delta x) - (\Delta v \cdot \Delta v) (\Delta v \cdot \Delta r - s^2)
\]

\[
\Delta v = (\Delta x, \Delta y) = (v_x - v_x', v_y - v_y') \\
\Delta v \cdot \Delta v = (\Delta v_x)^2 + (\Delta v_y)^2 \\
\Delta x = (\Delta x, \Delta y) = (x_i - x_f, y_i - y_f) \\
\Delta x \cdot \Delta x = (\Delta x_x)^2 + (\Delta x_y)^2 \\
\Delta v \cdot \Delta x = (\Delta v_x)(\Delta x_x) + (\Delta v_y)(\Delta x_y)
\]

**Important note:** This is physics, so we won't be testing you on it!
Particle-particle collision resolution

**Collision resolution.** When two particles collide, how does velocity change?

\[
\begin{align*}
\dot{v}_x^i &= v_x^i + J_x/m_i \\
\dot{v}_y^i &= v_y^i + J_y/m_i \\
\dot{v}_x^j &= v_x^j - J_x/m_j \\
\dot{v}_y^j &= v_y^j - J_y/m_j
\end{align*}
\]

Newton’s second law (momentum form)

\[
J_x = \frac{\Delta x}{s}, \quad J_y = \frac{\Delta y}{s}, \quad J = \frac{2m_im_j(\Delta v \cdot \Delta r)}{s(m_i + m_j)}
\]

Impulse due to normal force (conservation of energy, conservation of momentum)

**Important note:** This is physics, so we won’t be testing you on it!

**Event data type**

**Conventions.**

- Neither particle null ⇒ particle-particle collision.
- One particle null ⇒ particle-wall collision.
- Both particles null ⇒ redraw event.

```java
private static class Event implements Comparable<Event> {
    private final double time; // time of event
    private final Particle a, b; // particles involved in event
    private final int count; // collision counts of a and b

    public Event(double t, Particle a, Particle b) { ... } // create event

    public int compareTo(Event that) { return this.time - that.time; } // ordered by time

    public boolean isValid() { ... } // valid if no intervening collisions (compare collision counts)
}
```

**Particle data type skeleton**

```java
public class Particle {
    private double rx, ry; // position
    private double vx, vy; // velocity
    private double radius; // radius
    private final double mass; // mass
    private int count; // number of collisions

    public Particle(...) { ... }

    public void move(double dt) { ... }
    public void draw() { ... }

    public double timeToHit(Particle that) { ... } // predict collision with particle or wall
    public double timeToHitVerticalWall() { ... } // predict collision with vertical wall
    public double timeToHitHorizontalWall() { ... } // predict collision with horizontal wall

    public void bounceOff(Particle that) { ... } // resolve collision with particle or wall
    public void bounceOffVerticalWall() { ... } // resolve collision with vertical wall
    public void bounceOffHorizontalWall() { ... } // resolve collision with horizontal wall
}
```

**Collision system: event-driven simulation main loop**

**Initialization.**

- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

**Main loop.**

- Delete the impending event from PQ (min priority = 0).
- If the event has been invalidated, ignore it.
- Advance all particles to time $t$, on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.